A

22074661-25535

ASSISTANT COMMISSIONER FOR PATENTS
Washington, D.C. 20231

JC511 U.S. PTO
09/419749
10/16/99

Sir:

Transmitted herewith for filing is the patent application of Inventor(s) Tad Deffler and Eric Mintz

Entitled: **METHOD AND SYSTEM FOR EXTENSIBLE MACRO LANGUAGE**

Enclosed are:

☑ 27 sheets of specification and 2 sheet(s) of drawing(s).
☐ An unexecuted Assignment of the invention to Computer Associates Think, Inc.
☐ A certified copy of a priority application.
☐ A verified statement to establish small entity status under 37 C.F.R. §§ 1.9 and 1.27.
☑ An unexecuted declaration/power of attorney.
☐ An Information Disclosure Statement and form PTO-1449.
☐ Other

09419749 .101599

The filing fee has been calculated as shown below:

| | (Col. 1) | (Col. 2) | | SMALL | ENTITY | | OTHER THAN A SMALL ENTITY | |
|---|---|---|---|---|---|---|---|---|
| FOR: | NO. FILED | NO. EXTRA | | RATE | FEE | | RATE | FEE |
| BASIC FEE | | | | | $380 | OR | | $760 |
| TOTAL CLAIMS | -20 = | * 0 | | x 9 | $ | OR | x 18 | $ 0 |
| INDEP CLAIMS | -3 = | * 0 | | x 39 | $ | OR | x 78 | $ 0 |
| MULTIPLE DEPENDENT CLAIM PRESENTED | | | | x 130 | $ | OR | x 260 | $0 |
| * If the difference in Col. 1 is less than zero, enter "0" in Col. 2 | | | | TOTAL | $380 | OR | TOTAL | $760 |

☑ Please charge Deposit Account No. **02-0393** in the amount of **$ 760.00** to cover the filing fee. The Commissioner is also hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account No. **02-0393**. A duplicate copy of this sheet is enclosed.

☑ Any additional filing fees required under 37 C.F.R. § 1.16.
☑ Any patent application processing fees under 37 C.F.R. § 1.17.

☐ A check to cover the filing fee is enclosed.

☑ The Commissioner is hereby authorized to charge payment of the following fees during the pendency of this application or credit any overpayment to Deposit Account No. **02-0393**. A duplicate copy of this sheet is enclosed.

☑ Any patent application processing fees under 37 C.F.R. § 1.17.
☐ The issue fee set in 37 C.F.R. § 1.18 at or before mailing of the Notice of Allowance, pursuant to 37 C.F.R. § 1.311(b).
☑ Any filing fees under 37 C.F.R. § 1.16 for presentation of extra claims.

Respectfully Submitted,

Date: October 15, 1999

[illegible]

Eunhee Park (Reg. No. 42976)
BAKER & McKENZIE
805 THIRD AVENUE
NEW YORK, N.Y. 10022
(212) 751-5700

**EXPRESS MAIL NO.: EE22575057US**
**DATE OF DEPOSIT: October 15, 1999**

[NYC] 315713.1

22074661-25533

# *Application*

# *For*

# *United States Letters Patent*

**To all whom it may concern:**

Be it known that we,

Tad Deffler, and

Eric Mintz

have invented certain new and useful improvements in

METHOD AND SYSTEM FOR AN EXTENSIBLE MACRO LANGUAGE

of which the following is a full, clear and exact description:

Eunhee Park
Reg. No. 42,976
Baker & McKenzie
805 Third Avenue
New York, NY 10022

[Docket No. 22074661-25533]

# METHOD AND SYSTEM FOR AN EXTENSIBLE MACRO LANGUAGE

## CROSS-REFERENCE TO RELATED APPLICATIONS

The present application claims benefit of the filing date of U.S. Patent Application No. 60/104,682 entitled MODELING TOOL SYSTEMS AND METHODS, filed on October 16, 1998.

The present application is related to co-pending U.S. Patent Application No. _______ (Attorney Docket #22074661-25531) entitled METHOD FOR DETERMINING DIFFERENCES BETWEEN TWO OR MORE MODELS, being concurrently filed on the same day, which is incorporated by reference herein in its entirety.

The present application is related to a co-pending U.S. Patent Application No. _______ (Attorney Docket #22074661-25532) entitled METHOD FOR IMPACT ANALYSIS OF A MODEL, being concurrently filed on the same day, which is incorporated by reference herein in its entirety.

The present application is related to co-pending U.S. Patent Application No. _______ (Attorney Docket #22074661-25534) entitled METHOD AND APPARATUS FOR PROVIDING ACCESS TO A HIERARCHICAL DATA STORE THROUGH AN SQL INPUT, being concurrently filed on the same day, which is incorporated by reference herein in its entirety.

The present application is related to a co-pending U.S. Patent Application No. _______ (Atty. Docket #22074661-25535) entitled APPARATUS AND METHOD FOR MODELING TOOLS, being concurrently filed on the same day, which is incorporated by reference herein in its entirety.

## DESCRIPTION

## TECHNICAL FIELD OF THE INVENTION

The present invention relates in general to computer language processors and, particularly to an extensible macro language.

## BACKGROUND OF THE INVENTION

A macro is a set of commands that can be played back to perform a given task. Examples of these tasks include inserting a commonly used name and address into a word processor or executing a series of keystrokes to format a file. Tasks performed by macros are typically repetitive in nature allowing significant savings in time by executing the macro instead of manually repeating the commands.

Currently, different applications allow users to write macros and scripts within the confines of the allowed domain, i.e., within the confines of the specific application. For example, word processors typically allow users to create a macro by recording series of keystrokes to be played back later. Other applications allow users to create macros for retrieving and manipulating data within the confines of the applications. Thus, these applications include a limited set of macros, e.g., macro for recording keystrokes, a macro for retrieving data. The user is then typically limited to the macros provided by the application.

Frequently, however, each user using an application has a unique set of instructions or commands that the user would like to include as a macro in the application which was not previously provided. Because the macros are typically hard coded into the applications or into the macro language included in the applications, the only available method currently available to include additional macros into the application is to hard code the new macros into the application by modifying the source code and recompiling it before the new macro can be used. Usually, however, this presents a problem because the user is not given an access to the source code of the macro language or the application to modify. Moreover, it would be a tremendous burden on the application developers to try to cater to each individual user's preferences by customizing the applications to include the macros that the user would like to have.

Therefore, it is highly desirable to have an extensible macro language that would allow users to modify and extend the language to include their preferences when using the macro language. Furthermore, it is also highly desirable to be able to allow the users to extend the macro without having to modify or access the source code of the macro language since the source code is treated as a proprietary property not distributed to the users.

## SUMMARY OF THE INVENTION

To overcome the above shortcomings of the prior art macro language processors the present invention provides an extensible macro language that allows users to write new macro commands that include procedures tailored to the specific needs of the users without a need to modify any source code of the macro language processor. The extensible macro language is enabled to process the new macro commands by recognizing the new macro commands unknown to the language and associating the new macro commands with procedure calls stored in a registry, i.e., a repository, thereby allowing dynamic extension of a macro language.

In the present invention, a mechanism for dynamically registering new macro commands in a registry is also provided for allowing extensibility. To register new macro commands, the users may insert keywords representing the new macro commands and the associated codes or procedures in the registry for execution by the extensible macro language.

The present invention also defines a simplistic syntax for the extended macro language for recognizing the new macro commands for what they are without needing to know what functions they perform.

According to the goals of the present invention, there is provided a parser and a macro handler for processing macro commands not previously defined in the macro language. The macro commands not previously defined or undefined in the macro language refer to those macro commands that were not included in the set of commands available in the macro language at the time of release and distribution to the users. The parser analyzes keywords in a macro language expression and recognizes one or more keywords representing macro commands that were not previously defined in the macro language. The macro handler receives the keyword in the macro expression and retrieves from a registry of keywords, an executable code associated with the keyword. The executable code is run to process the macro command represented by the keyword. The registry of keywords may be augmented to include any keywords and associated codes for extending the macro language.

Further features and advantages of the present invention as well as the structure and operation of various embodiments of the present invention are described in detail below with reference to the accompanying drawings.

In the drawings, like reference numbers indicate identical or functionally similar elements.

## BRIEF DESCRIPTION OF THE DRAWINGS

Preferred embodiments of the present invention will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1 is a block diagram illustrating the components of the extensible macro language of the present invention; and

Figure 2 illustrates an example of a macro expression having an iterator macro.

## DETAILED DESCRIPTION OF THE INVENTION

The present invention is directed to an extensible macro language which may be extended dynamically in the runtime environment without having to rebuild and recompile the macro language. Although the extensible macro language may include a predetermined set of macro commands, the present invention allows users to add additional or new macro commands as desired. Figure 1 is a block diagram 100 illustrating the components of the system for providing the extensible macro language of the present invention. The parser 102 includes a grammar or syntax 104 that the parser 102 employs to analyze and parse a given expression. As shown in Figure 1, the parser 102 receives a macro language expression 106 and parses the expression into components according to the syntax 104 of the macro language. The syntax used in one embodiment of the present invention will be described in more detail hereinbelow. Referring back to Figure 1, the parser 102 reads the expression 106 recognizing certain tokens predefined in the syntax that indicate a presence of a new macro command. In this example, when the parser 102 encounters curly braces in the expression 106, parser 102 treats the keywords, for example, "property (name)", embedded within the braces as a new macro command. Moreover, the parser 102 recognizes, based on the syntax 104, that the "name" embedded within the parenthesis is a parameter to the new macro command. Other aspects of the syntax 104 may dictate that a string of characters outside any symbols to be interpreted as a literal string. Accordingly, the parser 102 breaks each element in the expression into components as shown at 108.

A novel feature of the parser 102 in the present invention is that the parser 102 is transparent to the actual content within the tokens, i.e., curly braces. That is, as long as the new macro commands or keywords are embedded within a recognizable token, the parser 102 breaks the keywords down into components regardless of whether the keywords have been predefined in the macro language. Thus, as shown at 108, the macro expression 106 is broken down into components according to the syntax 104 of the extended macro language. The new keyword "property" is broken down as a token component 108a; the string "name" within the parenthesis is broken down as a parameter component 108b; the string "likes" is broken down as a literal component 108c; and the string "pizza" is also broken down as a literal component 108d.

As shown in Figure 1, the present invention also includes a macro handler 110, and a repository 112 having keywords and their corresponding executable codes. The executable codes may be stored in the repository 112 as a pointer to the actual codes 114 for execution. The repository 112 includes one or more keywords and associated codes, and may be dynamically modified, e.g., new keywords and codes added to it as need arises by a user of the macro language. The repository 112 in the present invention may be a simple file with a table of keywords and associated codes. Alternatively, a separate database may be used as the repository 112.

After the macro expression has been parsed into separate components as described above with reference to the parser 102, the components are then passed to the macro handler 110 for additional processing. For the token component having the keyword "property" 108a, the macro handler checks a repository to the keyword "property". If found, the code associated with the keyword "property" is retrieved and executed. In executing the code, the macro handler 110 passes all the parameters found in the macro expression and parsed as parameters, to the executing code. The macro handler 110 does not need to know any other processing that may be performed inside the code itself. All that the macro handler 110 needs to recognize is that the "property" is a keyword to be looked up in the repository 112 for its corresponding code, and the specified corresponding code in the repository 112 to be executed with any parameters. The corresponding code is typically specified in the repository 112 as a pointer to the actual code itself 114.

After the proper execution of the code 114 specified in the repository, the macro handler 110 accepts one or more outputs, if any, of the executed code and places the outputs back into the macro expression in place of the keyword. Thus, in the example shown in Figure 1, the output of the code associated with the "property" with the parameter "name" may be MARY. Consequently, the result of the extended macro expression "{property (name)} likes pizza" at 106 is "Mary likes pizza" as shown at 116.

A novel feature of the present invention is that the macro handler, like the parser, need not know anything in the code or what type of functions are being performed by the executable code. The macro handler merely provides an initiation into the executable code that is associated with the keyword. In an exemplary embodiment of the present invention, it is left up to the users to define exactly what the code should do, and consequently, therefore, what command the keyword is to perform, thus providing a flexible and extensible macro language.

In the above example, the output MARY may have been obtained in various ways transparent to the macro language. For example, the name MARY may have been obtained by performing a search from the World Wide Web, or may have been obtained from a database using a query language, further illustrating the extensibility afforded by the present invention.

<u>The Language Syntax</u>

The syntax or the grammar employed in one embodiment of the extensible macro language will now be described in detail. The extensible macro language of the present invention includes a syntax (Figure 1 104) comprising literals, macros, comments and operator/scoping characters.

<u>Literal</u>

The syntax in this embodiment treats all text outside of curly braces as a literal, and is emitted exactly as typed. Within curly braces, text inside double quotes is treated as a literal. Such a scheme allows for embedding of a literal within macro calls. Some examples of a literal are illustrated as follows:

```
This text would be emitted just like this;
```

{"So would this text"}

Macros

Macros include instructions to the macro processor, like procedures or functions in a programming language. According to the syntax defined in the present invention, all macros are embedded within curly braces. In one embodiment, the macros fall into two categories: procedures macros and iterator macros.

Procedure macros are designed to perform some work. They may expand to a value; they may declare a variable; they may invoke a process. The actions performed are entirely specified by the designer of the macro. In one embodiment, the macros must, however, return a "true" value upon successful completion of their task and a "false" value upon failure.

The following expression illustrates a string literal, followed by a macro call for getting the page number when printing:

```
My Model Report - Page {HeaderPage}       : Input
My Model Report - Page 1                  : Output
```

In the above example, the HeaderPage is a macro defined by a user to extract a page number.

Iterator macros allow the user to traverse across data structures. Iterators are distinguished by the keywords "begin" and "end" that delimit a block of code following the iterator declaration. The code within the "begin/end" block is executed once for each iteration. When the iterator has moved across all objects in its pool, control breaks out of the iteration block and continues to execute a next statement in the macro expression after the block.

The following block of macro expression illustrates a use of the iterator macro:

```
{
    MyIterator
    begin
        DoSomething
    end
}
```

In the above example, the procedure macro "DoSomething" executes once for each element returned by the "MyIterator" macro. The curly braces surrounding the entire fragment indicates that all expression within the braces is to be treated as macro code.

<u>Parameters</u>

The syntax defined in the extensible macro language of the present invention allows for both procedure and iterator to accept and process parameters. Parameters may include strings, or other macros. To distinguish parameters, the parameters are enclosed within parenthesis following the macro. Macros may accept variable-length parameter lists, as desired. The following illustrates a macro expression having a parameter "foo":

```
{MacroWithParameters ("foo")}
```

Control blocks

In some instances, it is desirable to have a block of a macro expression to fail if any portion of it fails. The following example illustrates one such instance:

```
{FirstName [MiddleInitial "."] LastName}
```

If there was no middle initial, the MiddleInitial macro would return a nil value or a fail value. In that case, the literal "." should not be printed. To accommodate for such conditions, the present invention includes in its syntax, square brackets ("[]") that denote a conditional expression. Thus, if the macro within the square brackets fails, the rest of the expression in the square brackets is not emitted. In the above example, if the MiddleInitial failed for lack of value, the literal "." is not be printed.

The conditional blocks have internal scope, i.e., the failure of a conditional block does not affect the surrounding code. For conditions in a block to affect the outer block, the syntax additionally includes what is referred to as a propagating conditional denoted by angle brackets. If any macro within a pair of angle brackets fails, the block within the angle brackets as well as the next outer block fails. The following examples illustrate macro expression with a conditional and a propagating conditional:

```
{ Print " " [ Print [ Fail ] ] }      : input
foo foo                                 : output

{ Print " " [ Print < Fail > ] }      : input
foo                                     : output
```

In both examples the "Print" macro outputs the word "foo". In the first example, the failed macro in square brackets is contained within its block. Thus, the next outer block having "Print" is executed as well as the first "Print", resulting in the output "foo foo". In the second example, when a macro within angle brackets fails, the failure is propagated to the next block having the "Print" macro. Thus, the next outer block with "Print" is not executed. Since this Print macro is contained within a pair of square brackets, the failure is contained in the block. Thus, the first "Print" macro is executed, resulting in the output "foo".

Figure 2 illustrates an example of a macro expression including an iterator macro of the present invention. As described with reference to Figure 1, the keyword "ForEach" is recognized by the parser 102 (Figure 1) as a macro, and the word "Employee" is recognized as a parameter to the macro "ForEach". When the macro handler receives the token keyword "ForEach", the macro handler 110 (Figure 1) performs a look-up of the keyword "ForEach" in the registry 112 and executes the corresponding code. The code for "ForEach" macro, for example, may include instructions to perform commands found within the begin/end block of the macro expression for all sub-objects 204b, 204c in a given object 204 having the type of the specified parameter "employee". In this macro expression 202, another macro exists within the begin/end block. Accordingly, the macro handler 110 (Figure 1) performs a look-up of the keyword "Property" in the registry 112 and executes the corresponding code for each of the sub-objects 204b, 204c having employee type as specified in the "ForEach" keyword. The code associated with the "Property" keyword, for example, may include instructions to print the value of the type specified in the parameter of the keyword "Property", in this case, an employee name as specified by "EmpName". Consequently, the result of the macro expression 202 is the output shown at 208, "Mary John".

The extensible macro language of the present invention is useful for customizing macros specific to the needs of individual users. For example, the extensible macro language has been interfaced with the UMA Model for retrieving various objects from the UMA Model, as desired by a user. The UMA is disclosed in a co-pending U.S. Patent Application No. ______ (Atty. Docket #22074661-25535) entitled APPARATUS AND METHOD FOR MODELING TOOLS, filed on October 15, 1999, the disclosure of which is incorporated herein by reference in its entirety thereto. Appendix A includes a brief description of the extensible macro language of the present invention as used in the UMA Model and referred to as the UMA Template Language. The description in Appendix A explains one embodiment of the extensible macro language and should in no way be read as limiting the scope and capabilities of the extensible macro language to the descriptions contained therein.

While the invention has been particularly shown and described with respect to an embodiment thereof, it will be understood by those skilled in the art that the foregoing and other changes in form and details may be made therein without departing from the spirit and scope of the invention.

## CLAIMS

What is claimed is:

1. A method for providing an extensible macro language comprising:

analyzing a macro language expression;

determining based on a predetermined syntax of a macro language, one or more keywords in the analyzed macro language expression, the keyword representing a macro command not previously defined in the macro language;

retrieving a code associated with the keyword from a registry of keywords; and

executing the code associated with the keyword.

2. The method for providing an extensible macro language as claimed in claim 1, further comprising:

extending the registry of keywords by inserting a new keyword and a code associated with the new keyword.

3. A system for providing an extensible macro language, comprising:

a parser having a predefined syntax to determine one or more extended keywords embedded within a macro language expression, the extended keyword representing a macro command undefined in a predetermined set of macro commands of a macro language;

a keyword repository having one or more keywords and associated codes; and

a macro handler coupled to the parser for receiving the extended keyword from the parser, the macro handler in response to the received extended keyword, retrieving a code associated with the received extended keyword from the keyword repository and executing the code to run the macro command represented by the extended keyword.

4. The extensible macro language as claimed in claim 3, wherein the keyword repository is augmented to include new keywords and associated codes.

5. A method for parsing a macro language expression, comprising:

analyzing a macro language expression; and

determining based on a predetermined syntax of a macro language, one or more keywords in the analyzed macro language expression, the keywords representing macro commands undefined in a predetermined set of macro commands of a macro language.

# METHOD AND SYSTEM FOR AN EXTENSIBLE MACRO LANGUAGE

## ABSTRACT OF THE INVENTION

A method and system for an extensible macro language is provided. The system for providing the extensible macro language includes a parser and a macro handler for processing macro commands not previously defined in the macro language. The parser analyzes keywords in a macro language expression and recognizes one or more keywords representing macro commands that were not previously defined in the macro language. The macro handler receives the keyword in the macro expression and retrieves from a registry of keywords, an executable code or procedure associated with the keyword. The executable code is run to process the macro command represented by the keyword. The template language registry may be augmented to include any keywords and associated codes for extending the macro language.

1/2

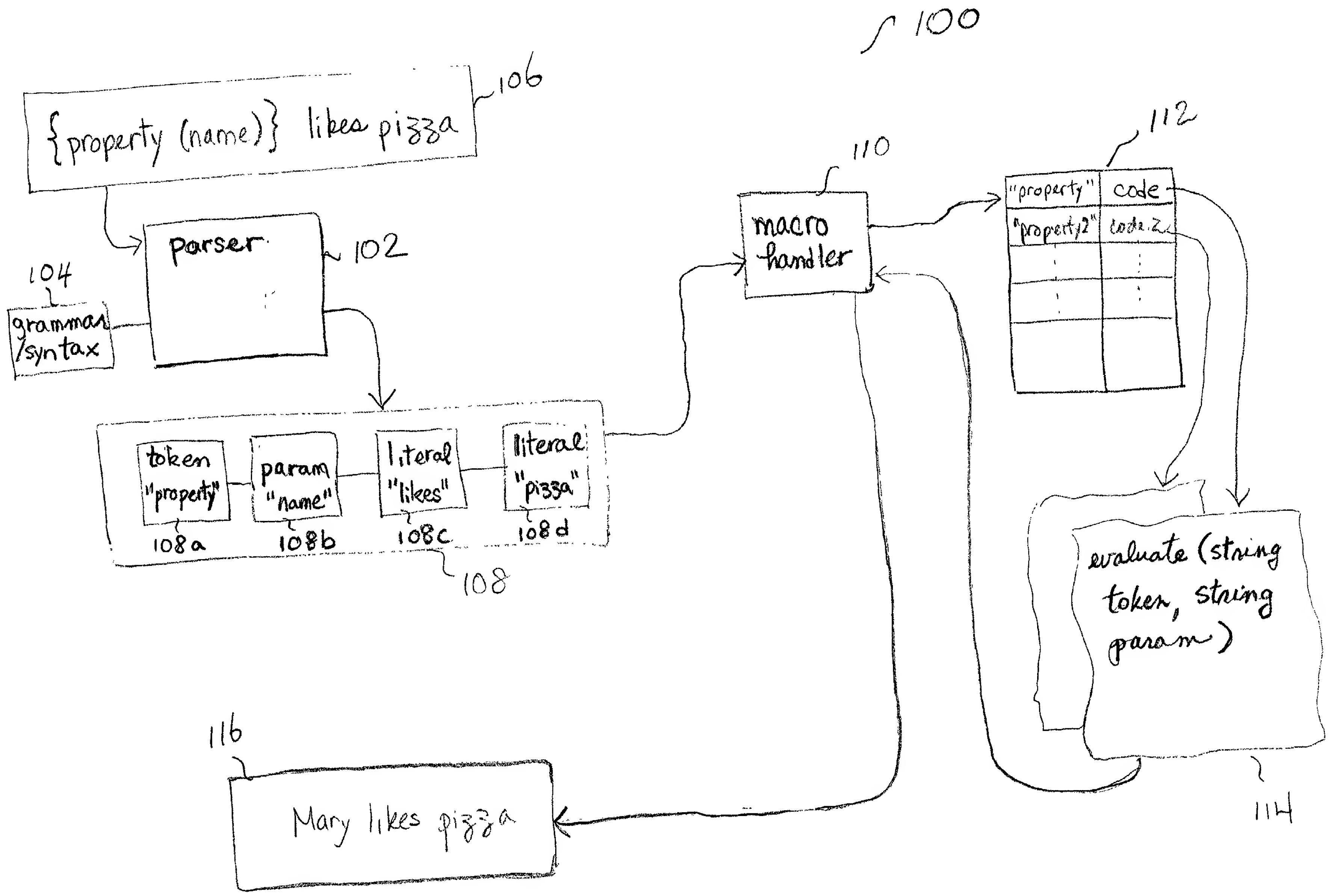
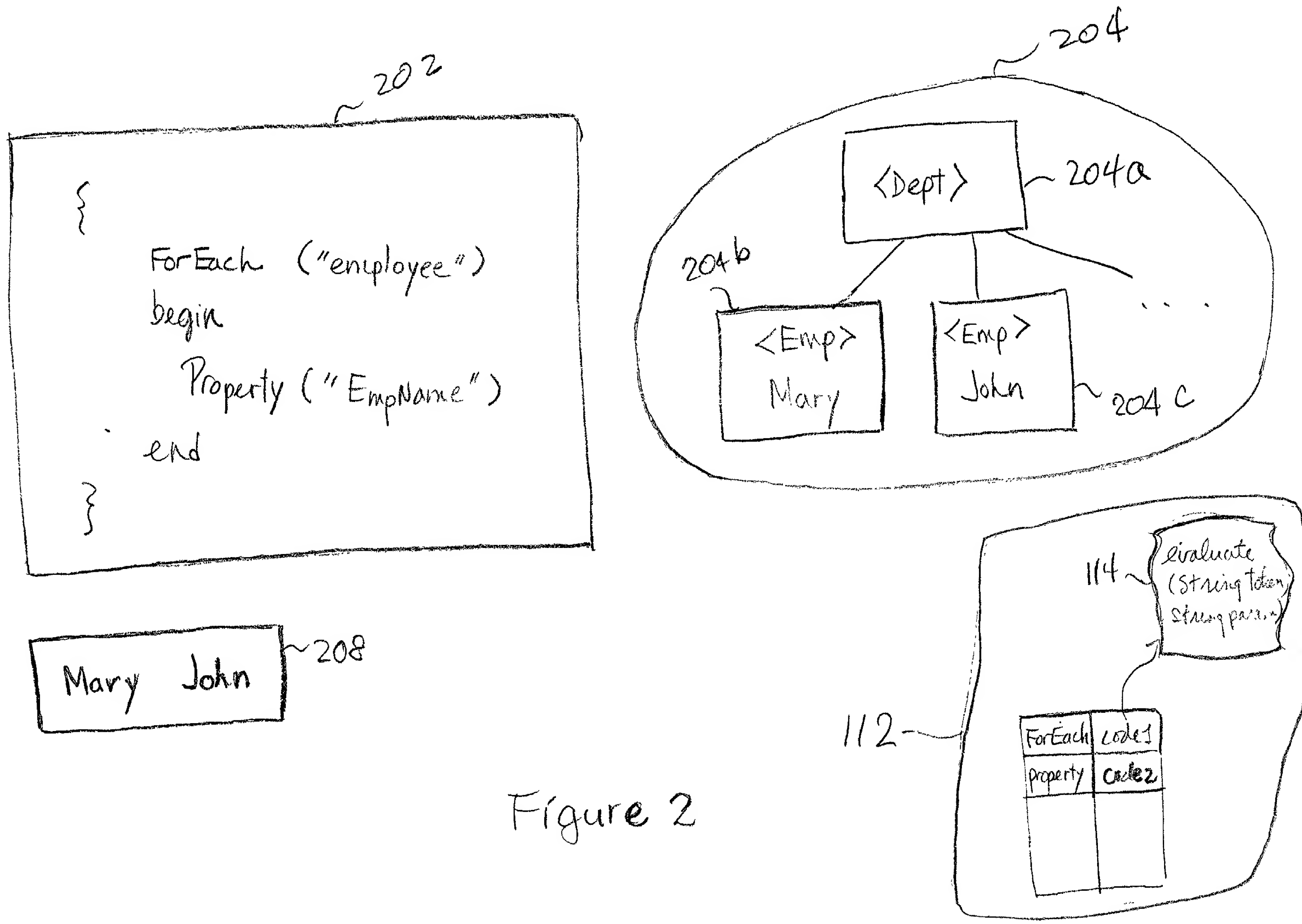


Figure 1

09419749.101699



```
{
    ForEach ("employee")
    begin
      Property ("EmpName")
    end
}
```

Figure 2

22074661-25535

# DECLARATION AND POWER OF ATTORNEY

As the below named inventor, we hereby declare that:

Our residences, post office addresses and citizenships are as stated below next to our names.

Tad A. Deffler
457 Rockaway Street
Boonton, New Jersey 07005

Eric Mintz
131 Woodbridge Avenue
Metuchen, New Jersey 08840

09475490 . 010500

We believe that we are the original, first and joint inventor of the subject matter which is claimed and for which a patent is sought on the invention entitled **METHOD AND SYSTEM FOR AN EXTENSIBLE MACRO LANGUAGE**, the specification which is being filed herewith.

We hereby state that we have reviewed and understand the contents of the above-identified specification, including the claims.

We acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, Code of Federal Regulations, § 1.56(a).

We hereby claim foreign priority benefits under Title 35, United States Code, § 119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application(s) for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

## PRIOR FOREIGN APPLICATION(S)

| Number | Country filed | Day/month/year | Priority Claimed Under 35 USC § 119 |
|---|---|---|---|

We hereby claim the benefit under Title 35, United States Code, §§ 119(e) of any United States provisional application(s) listed below:

**PRIOR PROVISIONAL APPLICATION(S)**

| Application Number | Filing Date |
|---|---|
| 60/104,682 | October 16, 1998 |

---

We hereby claim the benefit under Title 35, United States Code § 120, of any United States application(s) listed below or under § 365(c) of any PCT international application(s) designating the United States of America listed below, and insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States or PCT International application(s) in the manner provided by the first paragraph of 35 U.S.C. 112, I acknowledge the duty to disclose information which is material to patentability as defined in 37 CFR 1.56 which became available between the filing date of the prior application and the national or PCT international filing date of this application.

**PARENT APPLICATION(S)**

| U.S. Parent or PCT Parent Application Number | Parent Filing Date | Parent Patent Number (if applicable) |
|---|---|---|

---

And we hereby appoint James David Jacobs (Reg. No. 24,299), Victor DeVito (Reg. No. 36,325), Harry K. Ahn (Reg. No. 40,243), Frank Gasparo (Reg. No. 44,700) and Eunhee Park (Reg. No. 42,976) my attorneys with full power of substitution and revocation, to prosecute this application and to transact all business in the Patent and Trademark Office connected therewith.

Please address all communications regarding this application to:

BAKER & McKENZIE
805 Third Avenue
New York, New York 10022

Please direct all telephone calls to Eunhee Park at (212) 751-5700.

We hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful and false statements may jeopardize the validity of the application or any patent issued thereon.

First Inventor: Tad Deffler

Inventor's Signature: ______________________________

Date: ___________________

Residence: 457 Rockaway Street
Boonton, New Jersey 07005

Citizenship: USA

Post Office Address: Same as above.

Second Inventor: Eric Mintz

Inventor's Signature: ______________________________

Date: ___________________

Residence: 131 Woodbridge Avenue
Metuchen, New Jersey 08840

Citizenship: USA

Post Office Address: Same as above.

## Overview

The purpose of this document is to provide a brief introduction to the template language implemented in UMA. It assumes you have a working knowledge of the object/property metamodel of UMA.

It is not intended to provide an exhaustive explanation of the architecture. UMA team members can provide you with such an explanation and they can direct you to many code samples that use this facility.

The purpose of the UMA Template Language (UTL) is to provide scripting capabilities to any UMA-based application. These capabilities can be used for various features, including (but not limited to):

- **Macro expansion**: Runtime expansion of code templates against the model. Examples include trigger code and stored procedures in OR•Compass, and Visual Basic and Java components in SQL•Compass.
- **Scripting**: Execution of a series of commands against the model. An example is the controller code for Forward Engineering in OR•Compass.

The architecture of UTL is intended to promote its use in as many aspects of an UMA-based product as possible. This is accomplished by allowing the user[1] maximum flexibility in tailoring the language to the specific needs of his product, while providing as much implementation support as possible and exposing mechanisms that allow reuse of new implementation code.

To accomplish these goals, UTL:

1. defines a minimal syntax for the language;
2. allows the user to write procedures and interators;
3. provides a mechanism for registering the procedures and iterators in registries.

## Syntax Summary

The following is a brief summary of the syntax of UTL intended to provide the bare minimum necessary to understand the implementation. More information can be found in the documentation shipped with OR•Compass 1.0.

Template code consists of literals, macros, comments and a few operator/scoping characters. Currently, UTL views all code as strings: numeric values are described by their string representation; most macros evaluate to strings. etc.[2]

[1] In this document, "user" refers to the UMA application developer, not the end-user of the application.
[2] UTL 1.5, expected this fall, will also support numeric modes of operation that will provide better performance when dealing with numbers. This will be useful for features such as volumetric calculations.

## Literals

In template source code, all text outside of curly braces is treated as a literal, and is emitted exactly as typed. Within curly braces, text inside double quotes is treated as a literal. The former allows large blocks of boilerplate to be entered, e.g. the body of a stored procedure. The latter allows you to embed literals within macro calls.

*Example:*

```
      This text would emit just like this.
      {"So would this text."}
```

## Macros

Macros are special instructions to the macro processor...basically procedures. All macros are contained within curly braces. Macros basically fall into two categories: procedure macros and iterator macros.

Procedure macros are designed to perform some work. They may expand to a value; they may declare a variable; they may invoke a process. The action(s) performed are entirely specified by the designer of the macro. The only constraint upon them is that they return True upon successful completion of their task and False upon failure.

*Example—the following shows a string literal, followed by a macro call for getting the page number when printing:*

```
      My Model Report - Page {HeaderPage}      ⇦ Input
      My Model Report - Page 1                 ⇦ Output
```

Iterator macros allow the user to traverse across data structures. Iterators are distinguished by the keywords begin and end that delimit a block of code following the iterator declaration. All of the code within this begin/end block will be executed once for each iteration. When the iterator has moved across all objects in its pool, control will break out of the iteration block and execute the first statement after the block.

*Example—the following executes the procedure macro DoSomething once for each element returned by the MyIterator macro. Note the curly braces surrounding the entire code fragment...indicating that all code within is to be treated as macro code.*

```
      {
         MyIterator
         begin
            DoSomething
         end
      }
```

Macros, both procedure and iterator, can take parameters. Parameters are strings, or anything that evaluates to a string, such as another macro. If a macro takes parameters, they are enclosed in parentheses after the macro. Since forward declaration of macros is not required, macros may accept variable-length parameter lists, if desired.

*Example:*

```
{ MacroWithParameters("foo") }
```

### Control blocks

Occasionally, it is desirable to have an entire block of template code fail if any piece of it fails. An example would be where there was some boilerplate text followed by an optional value; if the value was not present, the boilerplate should not be emitted.

Any code enclosed in square brackets is *conditional*: it will all fail if any piece of it fails. "Failure" means that a macro will not execute and a literal will not be emitted.

*Example—if there is no middle name, the period will not emit:*

```
{ FirstName [MiddleInitial "."] LastName }
```

Conditional blocks hide their internal scope. This means that the failure of a conditional block has no effect on the surrounding code. Any code enclosed in angle brackets is *propagating conditional*: it will fail if any piece of it fails and it will propagate that failure out to the next block.

The best way to distinguish these two types of blocks is with examples. In both examples, the Print macro emits the word "foo" and the Fail macro fails no matter what.

*Example of conditional block:*

```
{ Print " " [ Print [ Fail ] ] }      ⇦ Input
foo foo                               ⇦ Output
```

*Example of propagating condition block:*

```
{ Print " " [ Print < Fail > ] }      ⇦ Input
foo                                   ⇦ Ouput
```

## Architecture

### LWMDataButler

The desired capabilities of UTL are achieved using an object called a data butler. Data butlers are responsible for providing implementations for the macros encountered by the macro processor when executing template code. When the macro processor is invoked, a data butler interface[3] is supplied to it. Essentially, a data butler is a use of the Strategy pattern.[4]

[3] Interface defined in LWMDataButler.h
[4] See Gamma, Helm, Johnson Vlissides, Design Patterns

This accomplishes the first goal of the language design. By providing different data butlers to the macro processor, the grammar of UTL can be configured for the application and task at hand. Each data butler implementation can accept a set of keywords (macros) appropriate for its task and reject keywords that are not acceptable. Further, a given keyword can be implemented in different ways by different data butlers, allowing template code to be ignorant of the underlying implementation. This is useful in allowing multiple products to work from the same templates when their underlying data models differ.

The LWMDataButler interface has five entry points:

```
// Handle procedure macro with no parameters
virtual LWTBoolean SubstituteValue(
            const LWTString & MacroName,
            LWTString & ExpansionValue) = 0 ;

// Handle procedure macro with parameters
virtual LWTBoolean SubstituteValue(
            const LWTString & MacroName,
            LWMStringList & ParameterList,
            LWTString & ExpansionValue ) = 0 ;

// Handle iterator macro with no parameters
virtual LWTBoolean StartIteration(
            const LWTString & IterName ) = 0 ;

// Handle iterator macro with parameters
virtual LWTBoolean StartIteration(
            const LWTString & IterName,
            LWMStringList & ParameterList ) = 0 ;

// Handle iterator macro iteration
virtual LWTBoolean NextIteration(
            const LWTString & IterName ) = 0 ;
```

### MCDataButlerl

The second design goal of UTL was to provide as much functionality as possible to the user and to allow them to publish their own code for reuse.

Since the metaphor of an UMA model deals with all data at a very abstract level—objects and properties—many grammatical elements of a template language can be written abstractly. These elements can be aggregated together to form the major portion of the language needed by any UMA-based product. The user could extend this base in the few areas that are product-specific. Ideally, the language should be extendable not only by the user, but by the end-user of the user's product. This is accomplished in our solution.

There are two basic problems that are addressed by the solution:

1. A mechanism is needed to share implementations. As new keywords are added, they must be published.

2. A mechanism is needed to control the context of a macro's execution. By this we refer to both the state of the UMA model at the time of execution, and to the specific object(s) or property(ies) the macro affects.

To meet these criteria, a base implementation of a data butler exists in UMA. This implementation, MCDataButlerI[5], provides mechanisms for defining the desired grammar incrementally and for managing contextual information in an UMA model.

This implementation is a superset of the LWMDataButler interface. Though the LWMDataButler interface is exposed, the user need not concern himself with it when implementing a data butler. The implementation in MCDataButlerI accomplishes all that has been found necessary to do in terms of implementing the LWMDataButler interface.

First, MCDataButlerI manages a registry of macro handlers. A macro handler is an object that knows how to perform a certain action, given a context. Conceptually, a macro handler is an example of a Command pattern.[6] The data butler maintains a dictionary of macro handlers keyed by macro name. When the macro processor requests the data butler to handle a macro (e.g. via SubstituteValue() ), the data butler locates the appropriate macro handler in its dictionary and delegates control to it. The macro handler processes the request and returns control to the data butler who simply forwards the return value (if any) and the success state to the macro processor.

A macro handler exposes an interface that is used by the data butler for invocation:

```
virtual LWTBoolean Execute(
            MCDataButlerI * Butler,
            const LWTString & MacroName,
            LWMStringList * Parameters,
            LWTString * ExpansionValue) = 0;
```

MCDataButlerI exposes an implementation signature for registering new handlers:

```
void AddHandler(
            const LWTString & Macro,
            MCMacroHandlerBase * Handler);
```

In order to implement a grammar, a user can construct an implementation of an MCDataButlerI and implement all required macro handlers. Alternatively, since the macro handler registry is a member variable of a data butler instead of a singleton, the user can subclass an existing data butler. This provides the user with all macros known to the subclass, plus those they develop. An example of this is the OR•Compass Forward Engineering data butler; it subclasses MCDataButler. The latter implements a wide variety of generic macros for creating objects, reading properties, iterating, constructing variables, etc. The Forward Engineering data butler implements one or two macros associated only with Forward Engineering (such as a macro for determining the correct

[5] Implementation found in MCDataButlerInterface.h
[6] *Op cit.*, Design Patterns

execution command for a database) and ends up with a complete grammar for emitting DDL.

Second, MCDataButlerI manages a context stack. The context stack allows a macro handler to know what object in the UMA model is being referenced when the macro executes. For example, the `Property()` macro retrieves a named property from an object. The macro handler for `Property()` needs to know which object it should query.

MCDataButlerI maintains a stack of object references and exposes an interface for pushing and popping objects on this stack, as well as for querying the stack. Macro handlers can query this interface to find out the context of their operation.

```
MCObject * GetCurrentContext(void);
void PopCurrentContext(void);
void PushCurrentContext(MCObject * Object);
```

Additionally, MCDataButlerI exposes a facility that allows MCObject iterators to manage the context automatically. Macro handlers for iterator macros can instantiate an iterator proxy. This is an object that automatically maintains the context stack for the iterator. Upon instantiation, it pushes the first object being iterated onto the context stack. Each subsequent iteration causes a pop and a push to occur. When there are no more items to traverse, the stack is popped, leaving it in the state that existed when the iterator was encountered. These iterator proxies are maintained in a stack of their own in order to manage nested iterators.

*Example—the following sample code shows how the Property macro would behave differently depending upon the context. Assume an entity named "MyTable1" that has an attribute "MyColumn1" and a second attribute "MyColumn2." Assume a second entity named "MyTable2" that has an attribute "MyColumn3." At the time this template code executes, we presume that the stack currently has the model on top due to some other code.*

```
{                                                   ⇦ Input
   ForEach( "UOEntities" )
   begin
      Property( "UPName" )
      ForEach( "UPAttributes" )
      begin
         Property( "UPName" )
      end
   end
}

MyTable1 MyColumn1 MyColumn2 MyTable2 MyColumn3 ⇦ Output
```